An empirical study of neural ordinal differential equations

Linjian Ma, Yufan Sun, Chiyu Ding
EECS Department, University of California, Berkeley
{linjian, yufan_sun, chiyuding}@berkeley.edu

Abstract

We perform an empirical study of the new family of deep neural network models, the neural networks based on Ordinary Differential Equations (ODE) solvers. The neural ODE method parameterizes the derivative of the hidden state using a neural network, instead of specifying a discrete sequence of hidden layers. In this report, we studied the background of neural ODE network and discussed the choice on the prevalent ODE implementation. We also experimented the ODE method on different datasets and analyzed the effect of ODE solver on neural network training. We experimentally verified the memory and model efficiency of neural ODE compared with the traditional ResNet and GRU model in image classification and time series prediction tasks, respectively. In addition, we demonstrate the ability of neural ODE in continuous function fitting. Our results reveal that this method can achieve similar and sometimes better accuracy than that the traditional neural network training procedure, but is also exposed to stability issues.

1 Introduction

We plan to perform an empirical study of the neural networks (NN) based on Ordinary Differential Equations (ODE) solvers. The idea of Neural ODE is proposed by (Chen et al., 2018) and is based on the previous studies that focus on the relation between neural networks and differential equations (Lu et al., 2017; Haber and Ruthotto, 2017).

The intuitions of neural ODE come as follows: for some efficient networks such as residual networks (He et al., 2016) and recurrent networks (Choi et al., 2015; Che et al., 2018), the transformations within the hidden units can be expressed by

$$\mathbf{z}_{t+1} = \mathbf{z}_t + f(\mathbf{z}_t, \theta_t),$$

where $\mathbf{z}_t$ denotes the hidden values at tth layer, and f can be regarded as the stack of many neighboring NN layers: $f = f_1 \circ \cdots \circ f_d$. From the perspective of numerical methods, the formulation above can also be regarded as an Euler discretization of a differential equation presented below, as more layers are added and smaller steps are taken:

$$\frac{d\mathbf{z}(t)}{dt} = f(\mathbf{z}(t), t, \theta).$$

Rather than using the traditional back-propagation to update the parameters in networks, once we view the forward pass as an ODE, we can leverage the adjoint method of ODEs and regard the back-propagation process as another ODE.

Defining neural networks based on ODE solvers has several advantages. First of all, rather than saving all the intermediate states of all the hidden layers as performed by the traditional back-propagation, ODE solvers are memory-efficient and don't require to save all the intermediate states. In addition, the Euler Method, which is leveraged by ResNet and Recurrent Networks only has first order accuracy and is not an efficient ODE solver. The state-of-art ODE solvers use adaptive methods and can produce higher accuracy, using which in the neural networks has larger potential.

We are interested in the effect of ODE solver on neural network training. Through empirical studies, we hope to answer the following questions:

- For the classification applicaitons, it has been shown that Neural ODE performs well on small datasets (MNIST (LeCun et al., 1998)). Is it still efficient for large datasets (e.g. Cifar10 (Krizhevsky and Hinton, 2009)), in terms of both accuracy and stability?

- Can Neural ODE be used to recover the functions parameterized by differential equations based on the its sampled data?
- It has been shown that Neural ODEs perform better than Recurrent Neural Networks on synthetic time-series data. Can it also perform well on application datasets?

Based on the three questions above, we plan to perform three sets of experiments as follows:

Classification models We will mainly focus on the Cifar10 Dataset, and try to compare the effect of neural ODE with Residual Networks. The empirical study will be used to test the effect of ODE solver in both memory efficiency and final accuracy efficiency. To show its memory efficiency, we will test the memory cost of the training process with different integration schemes. We will also investigate its effect on training stability as well as effects on final accuracy.

Function fitting We will test the ability of neural ODE in fitting the continuous functions parameterized by differential equations. We will test the cases when the input function is parameterized by linear and nonlinear ODEs, respectively.

Time-series prediction We will test the ability of Neural ODE in predicting blood glucose trajectories time series data (Fox et al. (2018)). We will compare the predictive root mean square error (RMSE) on test set for both Neural ODE and Recurrent Network and compare the differences.

2 Background of neural ODE backpropagation

As is explained in Chen et al. (2018), the major difficulty in using continuous-depth networks is performing the reverse-mode differentiation through the operations of the forward pass, because it will have high memory cost. We will testify this argument in our experiment section. The *adjoint method* introduced by Pontryagin (2018), which regards the parameter-update process of the neural ODE block as another ODE propagation, is more memory-efficient.

Consider optimizing a scalar-valued loss function $L()$ with the input being the result of an ODE solver:

$$L(\mathbf{z}(t_1)) = L\left(\mathbf{z}(t_0) + \int_{t_0}^{t_1} f(\mathbf{z}(t), t, \theta) dt\right) = L\left(\text{ODESolve}(\mathbf{z}(t_0), f, t_0, t_1, \theta)\right),$$

to perform gradient based optimization, we need both the *adjoint* $\mathbf{a}(t) = {\partial L}/{\partial \mathbf{z}(t)}$ and the derivative of the loss over parameters ${dL}/{d\theta}$. The dynamics of the adjoints are given by another ODE, which can be thought of as the instantaneous analog of the chain rule:

$$\frac{d\mathbf{a}(t)}{dt} = -\mathbf{a}(t)^{\mathsf{T}} \frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \mathbf{z}}.$$

It's worth pointing out that the expression above depends on $\mathbf{z}(t)$. Computing the gradients with respect to the parameters θ requires evaluating a third integral, which depends on both $\mathbf{z}(t)$ and $\mathbf{a}(t)$:

$$\frac{dL}{d\theta} = -\int_{t_1}^{t_0} \mathbf{a}(t)^{\mathsf{T}} \frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \theta} dt.$$

We can combine the two equations above with the equation for the evolution of $\mathbf{z}(t)$ and they will be used for gradient updates. Note that the vector-Jacobian products $\mathbf{a}(t)^T \frac{\partial f}{\partial \mathbf{z}}$ and $\mathbf{a}(t)^T \frac{\partial f}{\partial \theta}$ can be efficiently evaluated by automatic differentiation, at a time cost similar to that of evaluating f. All integrals for solving $\mathbf{z}$, $\mathbf{a}$ and $\frac{\partial L}{\partial \theta}$ can be computed in a single call to an ODE solver, which concatenates the original state, the adjoint, and the other partial derivatives into a single vector.

3 Background of numerical ODE

Differential equations involve derivatives of unknown solution function. In ordinary differential equations, all derivatives are with respect to single independent variable, often representing time. Numerical solution of differential equations is based on finite-dimensional approximation, and differential equations are replaced by algebraic equation whose solution approximates that of given differential equation.

For the differential equation

$$\frac{d\mathbf{z}(t)}{dt} = f(t, \mathbf{z}(t)), \quad \mathbf{z}(t_0) = \mathbf{z}_0,$$

Euler's method solves it through a one-step update and has local truncation error of $O(\Delta t^2)$:

$$\mathbf{z}_{n+1} = \mathbf{z}_n + \Delta t f(t_n, \mathbf{z}_n).$$

Midpoint method solves it through a one-step update and has local truncation error of $O(\Delta t^3)$:

$$\mathbf{z}_{n+1} = \mathbf{z}_n + \Delta t f(t_n + \frac{\Delta t}{2}, \mathbf{z}_n + \frac{\Delta t}{2}\mathbf{z}(t_n, \mathbf{z}_n)).$$

Classical Runge–Kutta method (*RK4*) solves it through a 4-step update and has local truncation error of $O(\Delta t^5)$:

$$\begin{aligned}
\mathbf{z}_{n+1} &= \mathbf{z}_n + \frac{1}{6}(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4), \\
\mathbf{k}_1 &= \Delta t f\Big(t_n, \mathbf{z}_n\Big), \\
\mathbf{k}_2 &= \Delta t f\left(t_n + \frac{\Delta t}{2}, \mathbf{z}_n + \frac{\mathbf{k}_1}{2}\right), \\
\mathbf{k}_3 &= \Delta t f\left(t_n + \frac{\Delta t}{2}, \mathbf{z}_n + \frac{\mathbf{k}_2}{2}\right), \\
\mathbf{k}_4 &= \Delta t f\Big(t_n + \Delta t, \mathbf{z}_n + \mathbf{k}_3\Big).
\end{aligned}$$

In addition, the adaptive Dormand-Prince method (*dopri5*) (Shampine (1986)) calculates 7 different slopes $\mathbf{k}_i, i \in \{1, \ldots, 7\}$ and these slopes are then used to find approximations of the next point such that the coefficients of the 5th-order approximate were chosen to minimize its error.

In our later experiment sections, we will test these four ODE update methods: *Euler, Midpoint, RK4, dopri5*.

4 Experiments on image classification

We perform experiments on the Cifar10 dataset, which is is one of the most popular benchmark equipment for classification applications. We test Neural ODE on it and compare the results with a Residual Network (ResNet20) classifier (He et al. (2016)). We implement our models with Pytorch based on the ODE solver open source package[1]. We use Stochastic Gradient Descent (SGD) to update the parameters, with the batch size set as 512, learning rate set to be 0.1, momentum set to be 0.95. We use weight decay (Krogh and Hertz (1992)) to regularize the training, whose value is set to be $5e-4$. We train the model for 150 epochs, and decay the learning rate at 60 and 120 epoch, with the ratio of 0.1. These parameters are chosen based on the suitable SGD parameter range (Shallue et al. (2018)). We change all the residual blocks in the ResNet into ODE blocks in the ODENet, with the expressions f same for both networks. For the ODENet, we tried 5 integration methods introduced in the Background Section. We summarize our results based on the accuracy comparison, memory comparison between ResNet and ODENet, and the stability issue of ODENet.

Accuracy Comparison The training curves are summarized in Figure 1. It can be observed that the final accuracy of ODENet will be higher with accurate integration method. Generally, the *dopri5* method is the most accurate, and *Euler* method is the least accurate one. The best ODENet training scheme (with *dopri5*) can achieve similar accuracy as, and even a little bit better final accuracy than the ResNet. However, it's worth pointing that training with *dopri5* is extremely slow (more than 10 times slower than ResNet based on our experiment with current implementations). Although ODENet has the potential to be faster after code optimization, the results partially show the bottleneck of ODENet.

Memory Comparison In general, ODENet with adjoint method needs less memory usage than ResNet. This is because ODE method doesn't need to store any intermediate quantities of the forward pass (e.g., for the function $f = f_1 \circ \cdots \circ f_d$, it doesn't need to store the results of f_i), which allows us to train our models with constant memory cost as a function of depth. In addition, there is potential to have less parameters in ODENet, which can slightly reduce the memory usage. When the hidden unit dynamics are parameterized as a continuous function of time, the parameters of nearby "layers" are automatically tied together, which reduces the number of parameters required.

However, it's difficult to see the memory saving based on our current implementation, because Pytorch automatically save the results of each layer f_i. Therefore, here we only compare the memory usage of ODENet with different integration methods, which is highly correlated. As shown in table 1, the GPU memory varies from over 1GB to 3GB

[1] `https://github.com/rtqichen/torchdiffeq`

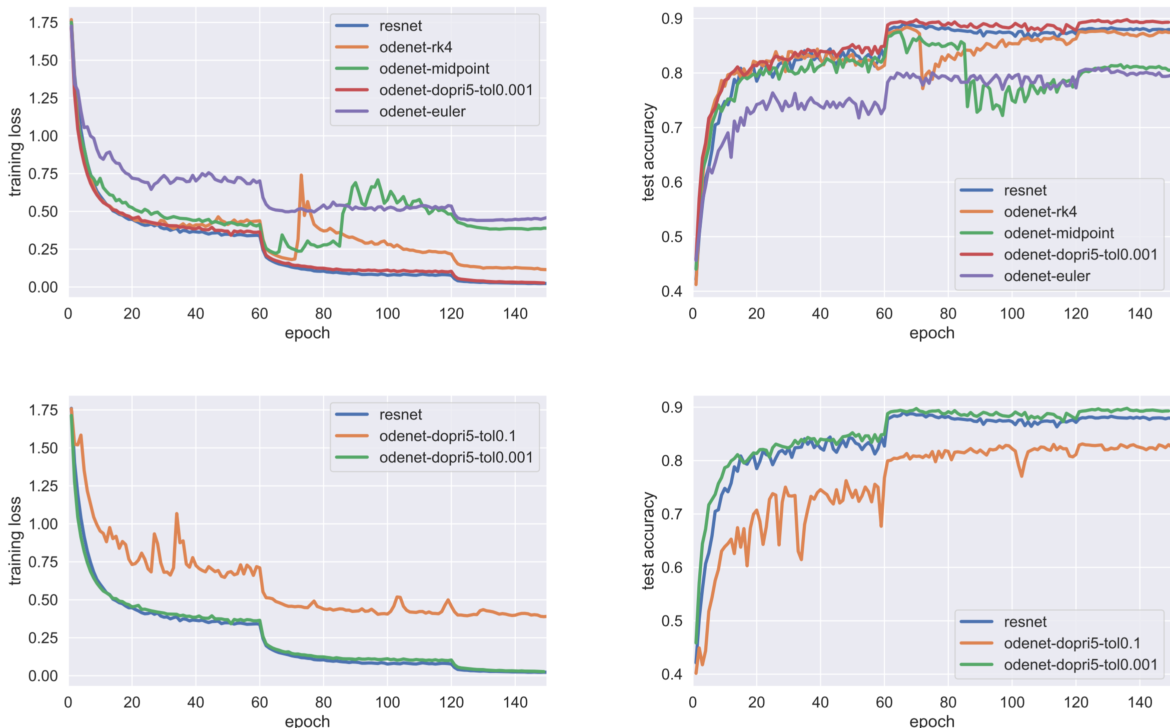

Figure 1: Performance comparison for image classification on CIFAR10. The upper two figures compare the ODENet with different integration method, and the bottom two figures compare the ODENet with *dopri5* method using different tolerance. Left: relation between training loss and epoch. Right: relation between test accuracy and epoch.

for different integration solvers. This is mainly because of the difference in number of iterations which will further affect the accuracy. To reach higher accuracy for numerical integration, more iterations are necessary based on the Background Section, and is therefore more memory-consuming. It therefore implies that for ODE method, we can trade precision with memory.

Stability Issue It can be seen from Figure 1 that training with neural ODEs is not that stable. There're points that the training loss / test error suddenly go up. It can be partially explained by the stability of ODE solutions. An ODE is called *stable* if solutions resulting from perturbations of initial value remain close to original solution. For a linear, homogeneous system of ODEs with constant coefficients has form

$$\frac{d\mathbf{z}(t)}{dt} = \mathbf{A}\mathbf{z}(t),$$

where $\mathbf{A} \in \mathbb{R}^{N \times N}$, when all the singular values of $\mathbf{A}$ denote $\lambda_i, i \in \{1, \ldots, N\}$, the solutions of the ODE are (asymptotically) stable if $Re(\lambda_i) \leq 0$ for $\forall i \in \{1, \ldots, N\}$, but unstable if $Re(\lambda_i) > 0$ for any eigenvalues. Although this linear ODE layer is simple, it can cover most of the linear layers (convolution, matrix multiplication) and ReLU layers in the neural network.

Suppose the forward propagation of an ODE block is stable. It means that all the eigenvalues of the matrix $\mathbf{A}$ are negative. However, when adjoint method is used to perform backpropagation, the ODE with $\frac{d\mathbf{z}(t)}{dt} = -\mathbf{A}\mathbf{z}(t)$ will be solved accordingly. As is discussed above, this ODE is unstable. This explanation can empirically explain the instability of ODE blocks with adjoint backpropagation: either forward or backward updates will involve the solution of an unstable ODE, making the training process of the whole neural network not stable.

We can see from Figure 1 that *dopri5* doesn't have the stability issue. Therefore, using accurate adaptive methods is always stabler than the accurate fixed-point methods (*RK4*), although it will result in more computations.

Table 1: Comparison of memory and number of forward/backward iterations for ODENet with different integration methods

Update method	dopri5 (0.001 tolerance)	dopri5 (0.1 tolerance)	rk4	midpoint	euler
GPU memory	3060MB	2534MB	1701MB	1598MB	1367MB
forward iterations	21	17	4	2	1
backward iterations	24	15	5	3	2

5 Experiments on function fitting

In this section we test the ability of Neural ODE to learn the true dynamics of some specific differential equations given sampled data. This is related to the inverse problem: given a set of observations, what's the ability to induce the factors under some uncertainties. To test this we will specify three ODEs, evolve it and sample points from their trajectories, and then restore it. These three ODEs are as follows:

Linear ODE First, we'll test a simple linear ODE. Dynamics is given as follows:

$$\frac{d\mathbf{z}}{dt} = \begin{bmatrix} -0.1 & -1.0 \\ 1.0 & -0.1 \end{bmatrix} \mathbf{z},$$

We will learn this dynamics using a one-layer linear network.

Spiral shaped nonlinear ODE We also test two spiral shaped nonlinear ODEs. The first dynamics is given as follows:

$$\frac{d\mathbf{z}}{dt} = \begin{bmatrix} -0.1 & 2.0 \\ -2.0 & -0.1 \end{bmatrix} \mathbf{z}^3,$$

We will learn this dynamics using an ODE block written as $\frac{d\mathbf{z}}{dt} = f_1 \circ f_2 \circ f_3 \, \mathbf{z}^3$, where f_1 and f_3 are two linear blocks, and f_2 is a tanh function.

The second nonlinear dynamics is given as follows:

$$\frac{d\mathbf{z}}{dt} = f(\mathbf{z}_0^T \mathbf{z}) \mathbf{A} (\mathbf{z} - \mathbf{z}_0) + \mathbf{B}(\mathbf{z} + \mathbf{z}_0) + f(-\mathbf{z}_0^T \mathbf{z}),$$

where $\mathbf{A} = \begin{bmatrix} -0.1 & -0.5 \\ 0.5 & -0.1 \end{bmatrix}$, $\mathbf{B} = \begin{bmatrix} 0.2 & 1 \\ -1 & 0.2 \end{bmatrix}$ and $\mathbf{z}_0 = \begin{bmatrix} -1 \\ 0 \end{bmatrix}$. We will learn this dynamics using a MLP network written as $\frac{d\mathbf{z}}{dt} = f_1 \circ f_2 \circ f_3 \circ f_4 \circ f_5 \, \mathbf{z}$, where f_1, f_3 and f_5 are three linear blocks, and f_2, f_4 are the nonlinear activation function (called ELU in PyTorch): $f(\mathbf{x}) = max(0, \mathbf{x}) + min(0, (exp(\mathbf{x}) - 1))$.

For each problem, we sampled 1000 points from the true dynamics, choose the update batch size to be 20, and run the experiments for 2000 epochs. We use Adam algorithm to optimize the parameters, and we choose the learning rate to be 0.01, the damping term to be 0.01. We choose the Mean Squared Error (MSE) as the loss function. For the fitting procedure, we tried *dopri5* and *RK4* integration methods. We didn't try other lower order methods because the accuracy of integration methods is important for the fitting of a continuous problem.

The final fitted trajectories are shown in Figure 2 and training curves are shown in Figure 3. It can be seen from the trajectory figure that the fitted results are close to the input dynamics. From the training curves, we can see that the training process will converge fast for linear problem, and will converge slower for nonlinear problems. Generally, *dopri5* is stabler than *RK4*, as can be seen from the bottom sub figure in Figure 3, which is consistent with our analysis in Section 4.

6 Experiments on latent time series prediction

In this section, we tested the neural ODE on predicting blood glucose trajectories time series data. The original data and model are borrowed from Fox et al. (2018). The aim is to predict the future blood goucose time series data based on the previous states, as is shown in Figure 4. One encoder is used to transform the input data $\mathbf{X}_{0:t}$ to the latent data $\mathbf{z}_t$, and recurrent units or their variations are used to produce the outputs $\mathbf{z}'_1, \cdots, \mathbf{z}'_h$ sequentially. In the reference, Gated Recurrent Unit (GRU) (Chung et al. (2014)) is used to generate these predictions, whose expressions are as follows:

$$\mathbf{z}_t = \sigma(\mathbf{U}_z \mathbf{h}_{t-1} + \mathbf{b}_z)$$
$$\mathbf{r}_t = \sigma(\mathbf{U}_r \mathbf{h}_{t-1} + \mathbf{b}_r)$$
$$\mathbf{h}_t = (1 - \mathbf{z}_t) \odot \tanh(\mathbf{U}_h (\mathbf{r}_t \odot \mathbf{h}_{t-1} + \mathbf{b}_h)) + \mathbf{z}_t \odot \mathbf{h}_{t-1},$$

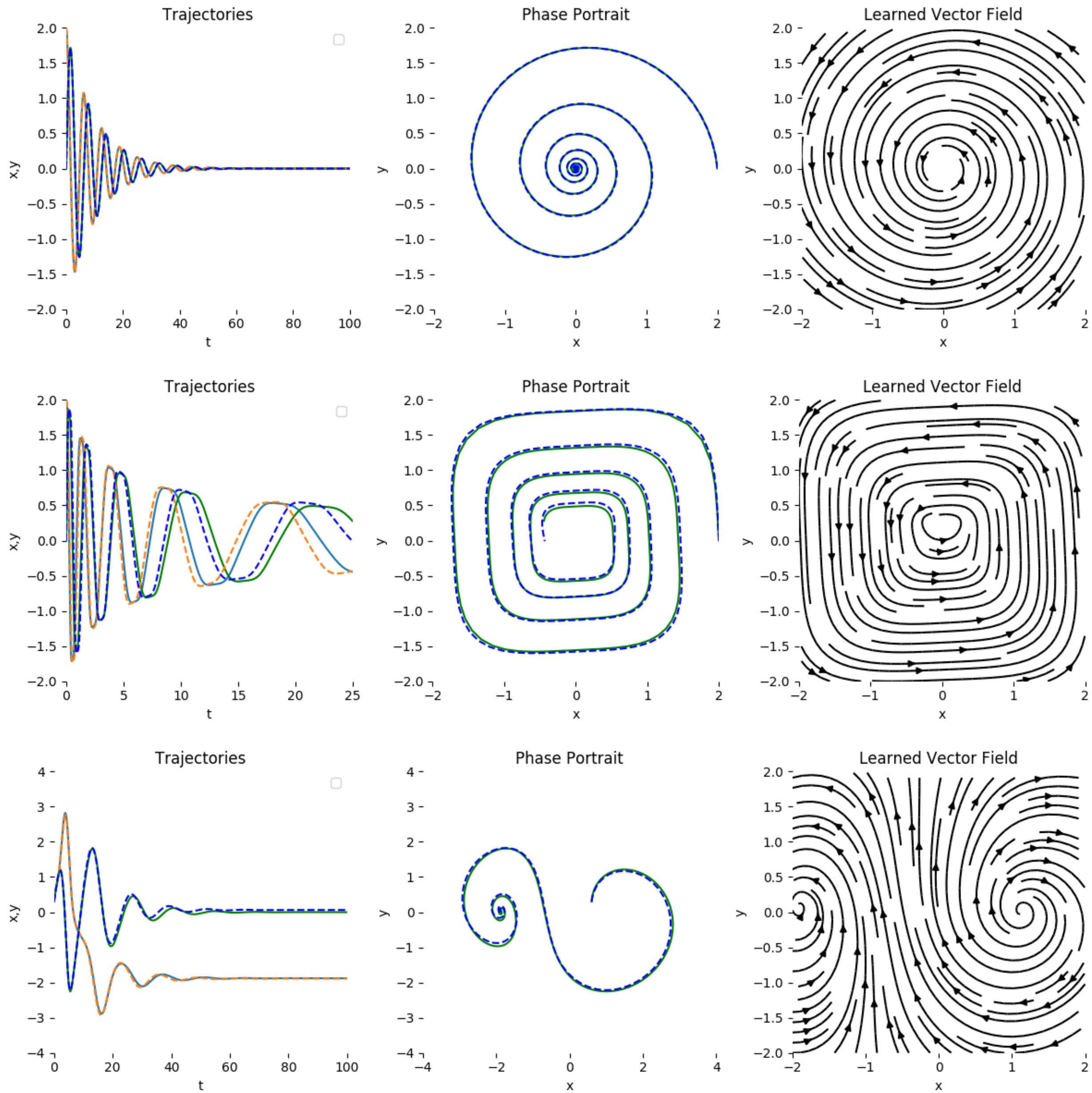

Figure 2: The trajectories, phase portrait and gradient vector field for the Linear ODE and two spiral shaped nonlinear ODEs introduced in Section 5. For the trajectories plots, the solid lines represent the true x and y coordinates with respect to time, and the dashed lines represent the fitted x and y coordinates. For the phase portrait, the relation between x and y under different times is shown, with solid line being true values and dashed line represents fitted ones.

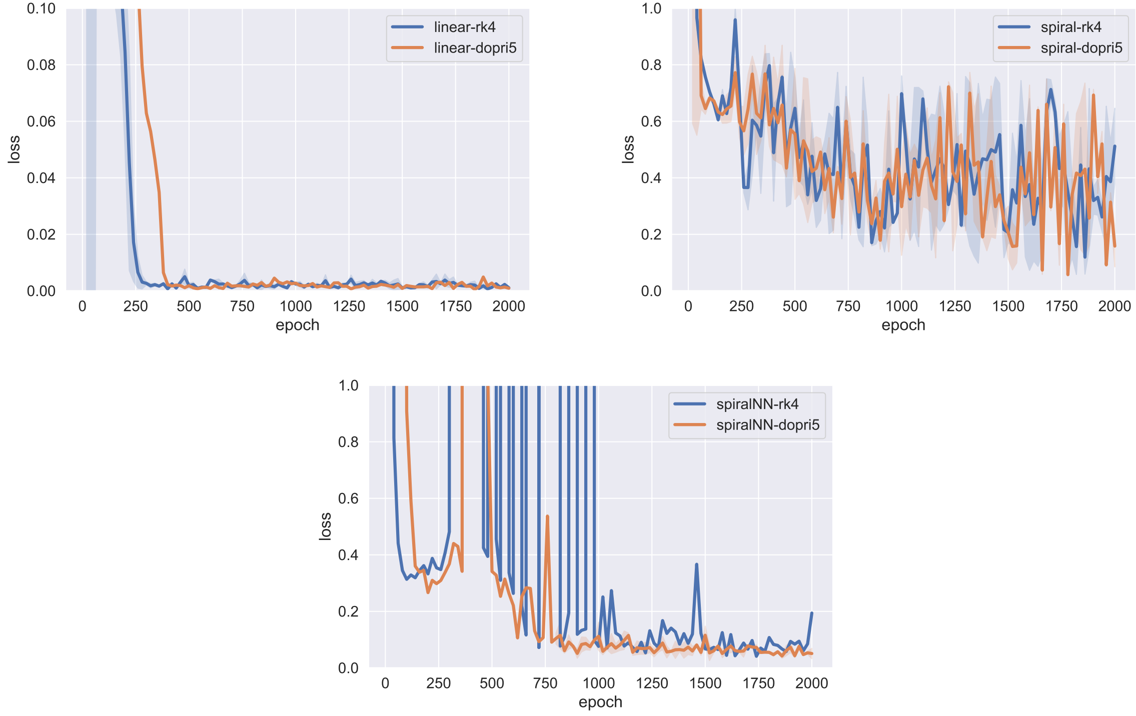

Figure 3: Loss values for different ODE integration methods with increasing epochs. Tested with linear ODE, spiral shaped ODE learned with three blocks, and spiral shaped ODE learned with MLP network.

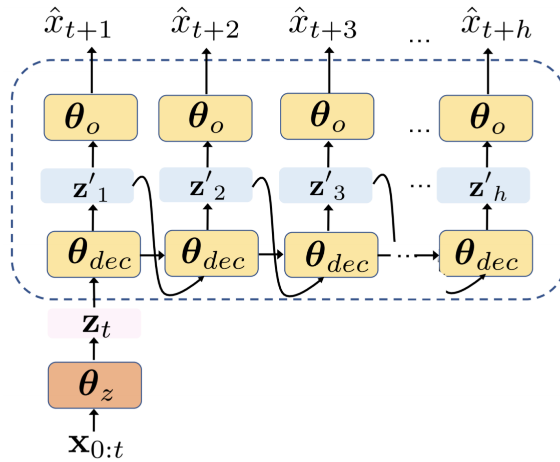

Figure 4: Network for blood glucose trajectories prediction in Fox et al. (2018), called "SeqMO". $\mathbf{X}_{0:t}$ represents the data from time 0 to time t, and $\hat{x}_{t+i}$ denotes the predicted data at time $t+i$. θ are parameters and both $\mathbf{Z}_t$ and $\mathbf{z}'$ are latent variables.

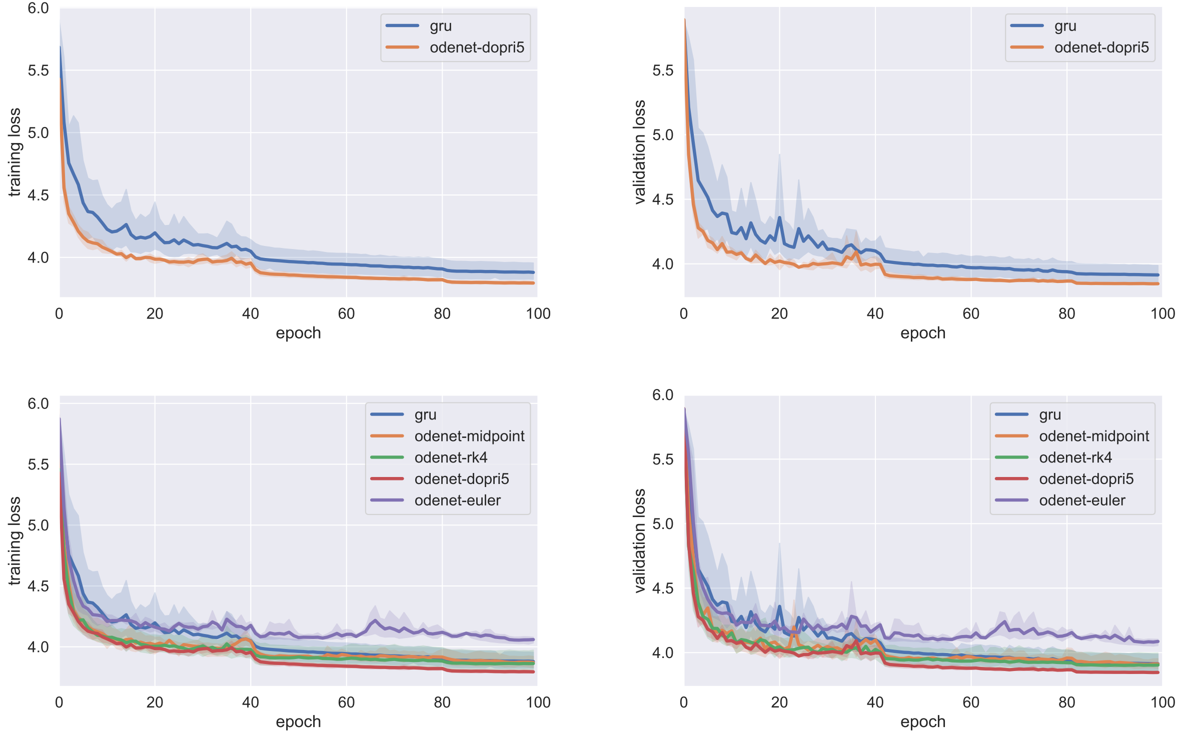

Figure 5: Loss comparison for latent time series prediction with increasing epochs. Left: training loss. Right: validation loss.

where $\mathbf{h}_t$ is an output state corresponding to $\mathbf{z}'_t$ in the figure. Based on the reference Jordan et al. (2018), the GRU expression above can be regarded as the *Euler* method of the following differential equation:

$$\mathbf{z}(t) = \sigma(\mathbf{U}_z \mathbf{h}(t) + \mathbf{b}_z)$$
$$\mathbf{r}(t) = \sigma(\mathbf{U}_r \mathbf{h}(t) + \mathbf{b}_r)$$
$$\frac{d\mathbf{h}(t)}{dt} = (\mathbf{z}(t) - 1) \odot (\mathbf{h}(t) - \tanh(\mathbf{U}_h(\mathbf{r}(t) \odot \mathbf{h}(t)) + \mathbf{b}_h)).$$

We tried to replace the GRU into the ODE block expressed above and test its effectiveness in the blood glucose example. Intuitively it makes sense: the time series data is continuous, and modeling that with continuous differential equations rather than the discrete GRU has the potential to be more accurate.

We implemented the model based on the implementations provided by the authors[2]. When testing the effectiveness of ODE block, we keep all the other parts as it is and only change the GRU. We test all the four integration methods introduced in the Background Section. For each experiment, we run it until 100 epochs, and choose the batch size 256. We use Adams algorithm to optimize the parameters, with the learning rate set to be 0.01 and the weight decay set to be $5e-4$. We choose the Mean Squared Error (MSE) as the loss function. The results are presented in Figure 5. We can see that with the *dopri5* integration method, ODE block has the potential to get smaller loss than the GRU. Except the *Euler* Method, all the other integration methods can achieve at least the same accuracy as GRU.

7 Conclusions

We perform an empirical study of the new family of deep neural network models, the neural networks based on Ordinary Differential Equations (ODE) solvers. We studied the background of neural ODE network and discussed the choice on the prevalent ODE implementation with different integration methods: *dopri5, RK4, Midpoint, Euler*. We also experimented the ODE method on three tasks: image classification, function fitting, and time series prediction. For

[2] https://github.com/igfox/multi-output-glucose-forecasting

the image classification task, we experimentally verified the memory and model efficiency of neural ODE compared with the traditional ResNet, and also show the weakness of neural ODE in stability. For the function fitting task, we show that neural ODE can be used to fit functions parameterized by both linear and nonlinear differential equations. For the time series prediction task, we demonstrate the ability of neural ODE in achieving better accuracy than the traditional recurrent units. Our future direction is to implement the neural ODE networks more efficient and investigate more theoretical problems.

8 Contribution

We calculate the team contributions based on three parts: experiment implementation (40%), report writing (30%), poster writing (20%) and other parts(include two check-ins and participation) (10%) .

Linjian Ma: experiment implementation (100%), report writing (50%), poster writing (20%), other parts (3.33%). Overall 60%.

Yufan Sun: report writing (25%), poster writing (40%), other part (3.33%). Overall 20%.

Chiyu Ding: report writing (25%), poster writing (40%), other part (3.33%). Overall 20%.

References

Zhengping Che, Sanjay Purushotham, Kyunghyun Cho, David Sontag, and Yan Liu. Recurrent neural networks for multivariate time series with missing values. *Scientific reports*, 8(1):6085, 2018.

Tian Qi Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. In *Advances in Neural Information Processing Systems*, pages 6572–6583, 2018.

Edward Choi, Mohammad Taha Bahadori, Andy Schuetz, Walter F Stewart, and Jimeng Sun. Doctor ai: Predicting clinical events via recurrent neural networks. *arXiv preprint arXiv:1511.05942*, 2015.

Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

Ian Fox, Lynn Ang, Mamta Jaiswal, Rodica Pop-Busui, and Jenna Wiens. Deep multi-output forecasting: Learning to accurately predict blood glucose trajectories. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1387–1395. ACM, 2018.

Eldad Haber and Lars Ruthotto. Stable architectures for deep neural networks. *Inverse Problems*, 34(1):014004, 2017.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

Ian D Jordan, Piotr Aleksander Sokol, and Il Memming Park. The expressive power of gated recurrent units as a continuous dynamical system. 2018.

Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009.

Anders Krogh and John A Hertz. A simple weight decay can improve generalization. In *Advances in neural information processing systems*, pages 950–957, 1992.

Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Yiping Lu, Aoxiao Zhong, Quanzheng Li, and Bin Dong. Beyond finite layer neural networks: Bridging deep architectures and numerical differential equations. *arXiv preprint arXiv:1710.10121*, 2017.

Lev Semenovich Pontryagin. *Mathematical theory of optimal processes*. Routledge, 2018.

Christopher J Shallue, Jaehoon Lee, Joe Antognini, Jascha Sohl-Dickstein, Roy Frostig, and George E Dahl. Measuring the effects of data parallelism on neural network training. *arXiv preprint arXiv:1811.03600*, 2018.

Lawrence F Shampine. Some practical runge-kutta formulas. *Mathematics of Computation*, 46(173):135–150, 1986.